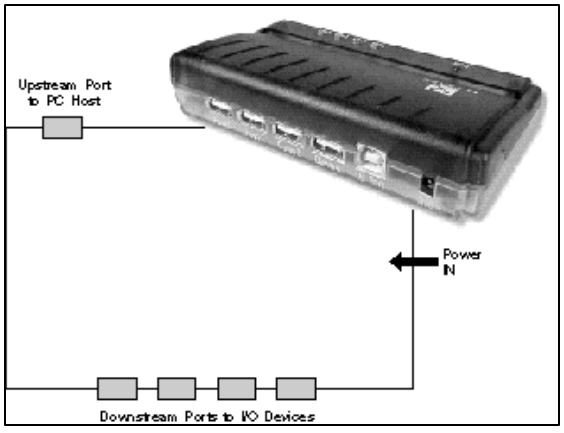
CHAPTER 14

HUBS – THE INSIDE STORY

The hub is USB's unsung hero. From the exterior view it looks like a simple splitter/combiner but in this chapter we look inside a hub to appreciate what it adds to a USB system solution. We shall see that the hub plays a major role in USB's hot plug and play features and in its managed power distribution. I have divided this chapter into three parts; the first presents a full/low-speed hub (1.1 compliant), the second presents a superset of this hub, a high/full/low-speed hub (2.0 compliant) and I conclude with an example design of a compound device – an I/O device with hub features. Note that we will NOT be designing a hub – these are available as single chip solutions and the USB specification gives you little design flexibility. It is more productive for you to buy a hub and to focus your energies on the I/O devices.

FULL/LOW-SPEED HUB

Figure 14-1 shows a representative hub product and its block diagram and this section will reveal what really goes on "under the hood."



Courtesy of Interex, Inc.

Figure 14-1. A hub provides connectivity and system power

THE BASIC HUB

We can identify three major hub elements: the hub repeater, the hub controller, and the port power control (Figure 14-2).

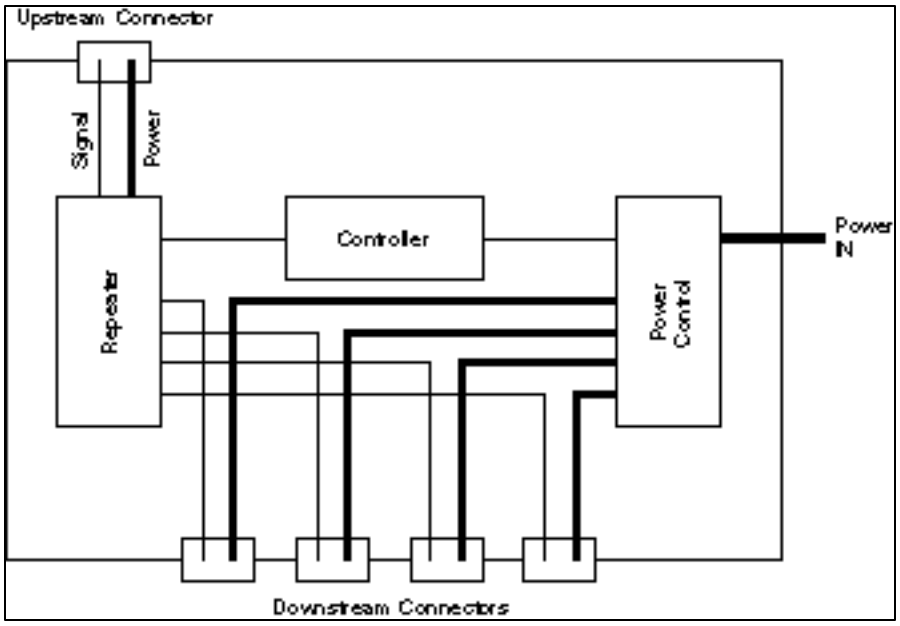


Figure 14-2. Identifying the major elements of a hub

Hub Repeater

The upstream port and the downstream ports can operate at 12 Mbps or 1.5 Mbps, depending on the capabilities of the I/O device currently being accessed. If no device is attached to a downstream port, it is disabled. To better understand hub operation, let's look at how signals pass through the hub. Let us assume, for the following discussion, that there are full-speed devices on ports 1 and 3, a low-speed device on port 2, and no device on port 4.

The hub decides on a packet-by-packet basis how to propagate signals between the upstream and downstream ports. Most packets arriving from the upstream port will be full-speed, and the repeater replicates these packets on all enabled, full-speed downstream ports. Some of these packets will initiate response packets from a device, and in the case of a full-speed device, a packet received on any of the downstream ports is replicated to the upstream port. The USB

protocol allows only one “talker”—this is typically the PC host but it will be the addressed slave device during the response phase. Therefore, the hub does not have to deal with multiple responses from many downstream ports because this is not permitted.

For full-speed packets, the hub broadcasts downstream packets to all enabled downstream ports and routes response packets from one of the downstream ports to the upstream port. This is summarized in Figure 14-3.

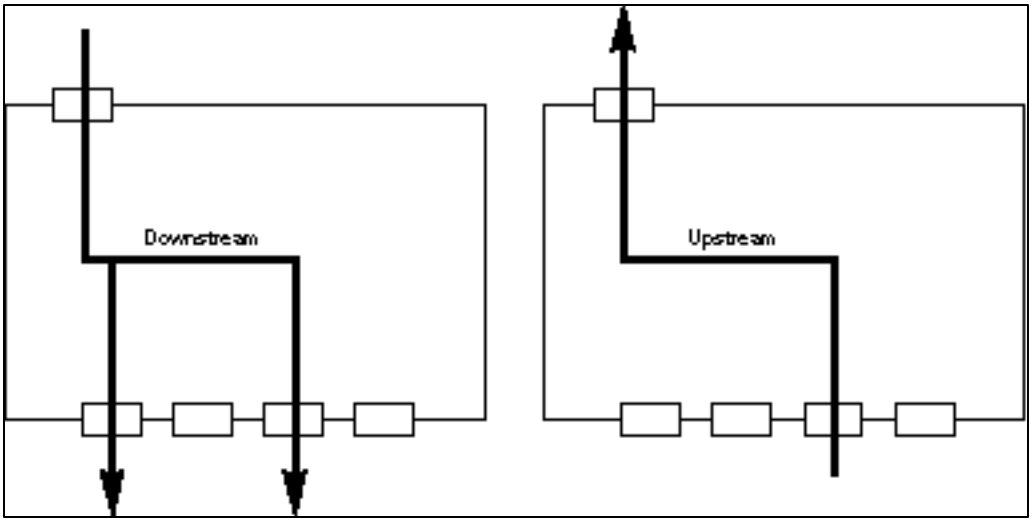


Figure 14-3. Downstream packets are broadcast, upstream packets are routed

Low-speed I/O devices get special treatment. Packets destined for low-speed I/O devices are sent from the PC host with a preamble token, which alerts the hub that the following packet is low-speed. The hub will replicate the preamble token and the subsequent low-speed packet on all enabled full-speed downstream ports and will propagate the low-speed packet, without the preamble, onto enabled low-speed ports. Only low-speed packets are propagated on low-speed downstream ports because of the filtering action of the hub. The PC host will always send at least one packet to all low-speed downstream ports every frame, to prevent them from suspending when there is no other low-speed bus traffic.

A low-speed I/O device will, of course, respond at low speed. The hub replicates this response on the upstream port.

The operation of the hub repeater is autonomous and does not need the assistance of the hub controller. The hub controller’s role is to manage system configuration changes.

Hub Controller

A hub is an I/O device. When first attached to a PC host, it is enumerated like any other USB device. Figure 14-4 shows the descriptors that a basic hub will present to the PC host during enumeration. The host software discovers that this is a hub by the Class code (byte at offset 5). A basic hub has one configuration and one interface, and it requires an interrupt endpoint 1 to report status changes. The hub class of devices includes a HUB descriptor.

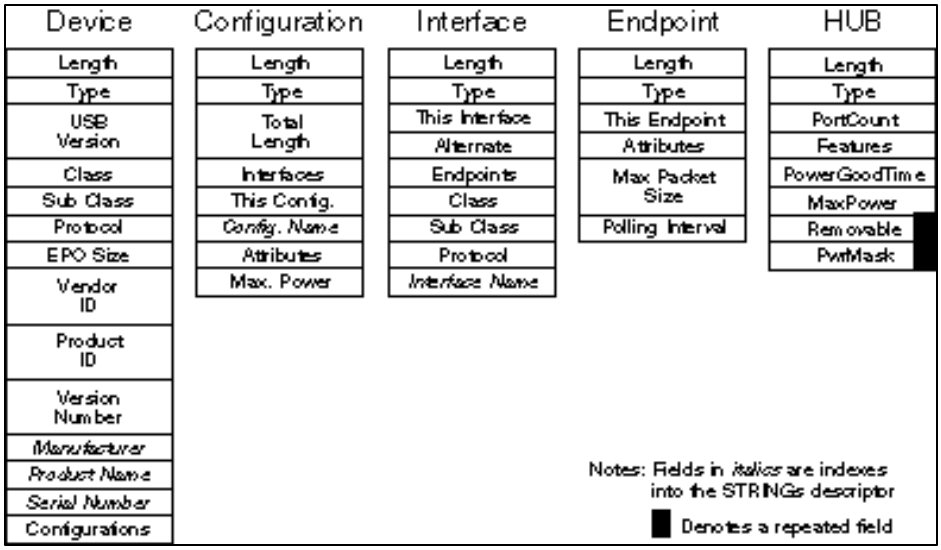


Figure 14-4. Descriptors for a basic hub

We have seen device, configuration, interface, and endpoint descriptors before (Chapter 3), and a hub uses the same definitions. The new HUB descriptor has the following entries.

A hub descriptor has a variable **Length** depending on the number of downstream ports that it is supporting.

The **Type** field is always 9.

The **PortCount** field identifies the number of downstream ports that this hub supports. This will include embedded I/O functions.

The **Features** field is bit-mapped as shown in Figure 14-5.

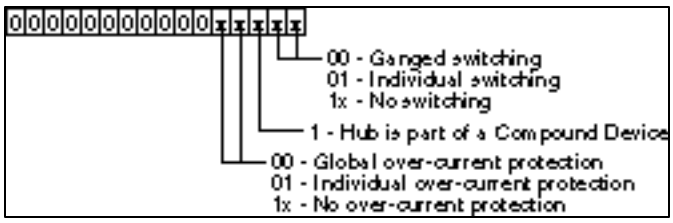


Figure 14-5. Definition of Features bit fields

The time that software should wait after power is applied to a hub is entered in the **PwrGoodTime** field in 2-ms increments.

The **MaxPower** field indicates the maximum current that the hub electronics will consume.

Removable is a bit field that indicates if a device on this port can be removed. Later in this chapter you'll find that this field is important when we add embedded I/O devices to the basic hub. Embedded devices are not removable, and the system software can make some operational optimizations.

PwrMask is also a bit field that indicates if the port has independently switchable power.

As a hub-class device, the hub controller must accept hub-class requests in addition to standard requests. For the full list of requests that must be serviced, see Table 14-1.

Table 14-1. Requests that a hub-class device receives

Type	Request	Our Action
Standard	Get_Status	Return current status
Standard	Clear_Feature	Clear Specified Feature
Standard	Set_Feature	Set Specified Feature
Standard	Set_Address	Store Unique USB Address and use from now on
Standard	Get_Descriptor	Return requested descriptor
Standard	Set_Descriptor	Optional: Set Specified Descriptor
Standard	Get_Configuration	Return Current Configuration or 0 if not configured
Standard	Set_Configuration	Set Configuration to the one specified
Standard	Get_Interface	Optional: Return Current Interface
Standard	Set_Interface	Optional: Set Interface to the one specified
Standard	Sync_Frame	Optional: Synchronize USB Frame Numbers
HUB Class	Get_Bus_State	Optional (for diagnostics): Return D+, D–
HUB Class	Get_Hub_Status	Return Hub Status with Changed Identified
HUB Class	Get_Hub_Descriptor	Return Hub Descriptor
HUB Class	Set_Hub_Descriptor	Optional: Set Hub Descriptor
HUB Class	Set_Hub_Feature	Enable a standard hub feature
HUB Class	Clear_Hub_Feature	Disable a standard hub feature
HUB Class	Get_Port_Status	Return Port Status with Changed Identified
HUB Class	Set_Port_Feature	Enable a standard port feature
HUB Class	Clear_Port_Feature	Disable a standard port feature

Power Control

Port power control consists of overcurrent protection, power-switching, or a combination of the two. Power can be supplied to all downstream ports in parallel or individually. It is cheaper to supply power to all downstream ports in parallel because fewer devices are required, but an overcurrent fault on one port will cause the other ports, in the same hub, to trip also—not a good user experience. Individually powered ports would also include separate overcurrent indicators that can be supplied to the operating software so that specific error messages can be presented to the user.

It is possible to derive the output power for the downstream ports from the upstream bus port. This would be called a bus-powered hub, and I do not recommend this approach. A total of 500 mA is available from a USB port, so a bus-powered hub would need to redistribute this power to its downstream ports. The hub electronics use 100 mA and could provide only 100 mA for each of four downstream ports. If an I/O device that needed more than 100 mA for operation were attached to this bus-powered hub port, it would be enumerated but not enabled. This could confuse many users. I recommend self-powered hubs, which means the hub has a separate power-in connector, so that 500 mA is always available for the downstream ports. The Windows operating system will alert a user if a high-power device is attached to a low-power hub socket and will recommend an alternate attachment point if possible.

The USB Specification defines the maximum amount of voltage drop and minimum acceptable output voltages. For bus-powered hubs, the industry has responded by supplying power switch devices with low series resistance, devices such as the Texas Instruments TPS2014/5 and the Micrel MIC2525 .

To meet the voltage drop, droop, and EMI requirements in the USB Specification 2.0, careful PCB layout is necessary. The following guidelines must be considered:

Keep all Vbus traces as short as possible and use at least 50-mil, 1-ounce copper for all Vbus traces.

- Avoid vias as much as possible. If vias are necessary, make them as large as feasible.
- Place the output capacitor and ferrite beads as close to the USB connectors as possible.
- Use a separate ground and power planes if possible.

The USB Specification also defines overcurrent protection for the downstream ports. Overcurrent protection is required on self-powered hubs and may be implemented with a resettable PTC such as Raychem's miniSMDC150 or a power distribution switch such as Texas Instruments TPS2014/5 (application information is included, with permission, in the Chapter 14 directory). Some hubs, such as the Philips ISP1122, integrate individual power control and the overcurrent protection. Power control is performed using a pMOS transistor on each port, and the "ON" resistance (Rds) of the transistors is used to trigger an overcurrent condition when too much current passes through them. The Philips ISP1122 reference design (included in the Chapter 14/Philips directory on the CD-ROM) details this feature in their Figure 8. Figure 14-6 shows a typical downstream port within a hub.

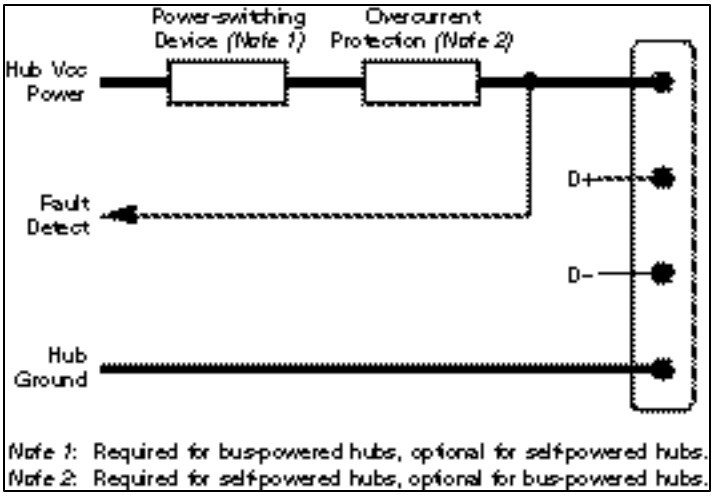


Figure 14-6. Power management of a downstream port

Basic Hub Summary

The operation of a basic hub is well defined by the USB Specification. Many implementations exist (see Appendix A) that are broadly categorized as hard-logic controlled or microprogrammed. Hard-logic devices, such as the Texas Instruments TUSB2043 and the Philips ISP1122, operate as a hub with no additional effort. You just "connect the dots" with careful PCB layout as recommended by their reference design, and you have a working hub.

A microprogrammed device, such as the Intel 8x930HX or the STMicroelectronics ST92161, includes a microcontroller that executes a program to implement the hub controller functions. The hub repeater functions are hard-wired in these devices to meet the timing constraints of the USB Specification. The manufacturers of these devices supply prewritten code for a hub design, so this approach is also connect-the-dots for a working hub.

The design freedom comes in the power controller, but I recommend self-powered with individually controlled outputs.

A full/low speed USB segment operates mainly at 12Mb/s and "downshifts" to 1.5Mb/s when communicating to a low-speed device. Transferring 8 bytes of data from a low-speed device takes approximately the same time as transferring 64 bytes of data from a full speed device due to this speed shifting. When multiple devices are connected on the downstream ports (the typical case) the upstream bandwidth is shared by these multiple devices. Hubs can be connected together to form a tree that can be up to six levels deep from the PC host.

HIGH/FULL/LOW-SPEED HUB

Version 2.0 of the USB Specification added a high speed of 480Mb/s and, as we have seen in previous chapters, this is of little consequence to the I/O device designer. Yes, higher speed transceivers and SIE's must be used but the user model and the software is THE SAME. Much of the responsibility for supporting this seamless upgrade model is borne by the USB 2.0 hub whose complexity is increased. Again, I do not intend that you design a 480Mb hub product (this is a daunting task) but, by exposing its operation, you can design better I/O devices and configure better USB system solutions.

A high/full/low-speed hub is a superset of a full/low-speed hub and its bandwidth allocation is superior. Figure 14.7 shows a block diagram of a high/full/low-speed hub and a representative product implementation.

Figure 14.7 Block diagram of high/full/low-speed hub

Notice that I have used the term “Mini Host Controller” rather than the USB specification terminology of “Transaction Translator”. Its **function** is to operate as a host controller supporting full/low-speed devices in a high-speed environment; it **implements** this by translating transactions sent to it by the PC host’s controller. I think that by calling it a mini host controller makes it easier to understand what it is doing.

A high/full/low-speed hub operates in one of two modes depending upon the connection speed of the upstream port. If this hub is connected to a full/low-speed segment then it operates exactly as a full/low-speed hub. The high-speed circuitry is disabled and its operation is as described in the previous section. This may seem like a waste of capability but it does provide a simple user model – you can plug an upstream cable into any downstream port in a tier above (i.e. closer to the PC host) you. This is the same paradigm that USB has used from its introduction. The operating system may warn you that this connection is sub-optimal from a system efficiency perspective, but everything will work as expected.

If a high/full/low-speed hub is connected to a high-speed USB segment then the full/low-speed circuitry is powered down and the high-speed units take over. It is easier to explain the operation using an example so, in Figure 14.7 assume that the following connections are made; 1 = high-speed, 2 = full-speed, 3 = low-speed, 4 = full-speed.

The operating system’s USB device driver knows the operating speed of each I/O device – it discovered this during the enumeration stage. This is important since it will take different actions depending upon the connection tree that your I/O device built during enumeration. Your application software does not need to know this level of detail and it will operate unchanged.

A new USBD.SYS is required to operate with new 480Mb/s host controllers – this is currently available, in beta form, for Windows 2000 operating system and it’s position in the operating system stack is shown in Figure 14-8.

Communicating with the high-speed I/O device on downstream port 1 is, in fact, the simplest case. The high-speed packets from the upstream port will be synchronized and re-clocked through the high-speed repeater and routed to port 1 (in reality there is more complexity than this due to the high-speed of the signal but all this detail is handled within the hub component and is not viewable outside of the device). Similarly, high-speed responses from the downstream port are re-synchronized and reclocked through the high-speed repeater and transmitted out of the upstream port.

Figure 14.8 Windows 2000 contains a new hub driver.

Communicating with a full-speed I/O device is a little more complicated. Note, that you need not know HOW this actually operates since the RESULT is the same from the I/O devices perspective: an application program issues, say, a read request and a read token is delivered to the correct I/O device. It was considered a HUGE waste of system bandwidth to downshift the bus speed from 480 to 12 (or, worse still 480 to 1.5) so the high-speed hub uses a "store and forward" technique which maintains the high-speed segment always at 480Mb/s.

To implement a READ of the full-speed I/O device the PC host controller sends a "read-request" (the USB specification calls this a start split) to the Mini Host Controller. This request is shown in Figure 14.9 – note that it contains the address of this particular mini-host controller, the specific port that the I/O device is on and the address of the targeted I/O device.

Figure 14.9 A "read-request" is issued.

The mini host controller absorbs this at 480Mb/s and generates an ACK. The mini host controller will then initiate the read on the full-speed bus and the I/O device will respond as normal (i.e. provide data, NAK or STALL). The mini host controller stores this response locally. In practice the mini host controller will be managing several outstanding requests on full and low-speed segments due to the huge speed difference between them and the high-speed segment therefore the mini host controller will have many buffers.

Some time later the PC host controller will send a “read-done-yet-request” (the USB specification calls this a complete-split and it contains the same addressing information so that the mini host controller can match this request with the initial start split). If the mini host controller has not received a response to the initial read request yet then it will respond with a NAK, else it will forward the response provided by the I/O device. This sequence is shown in Figure 14.10. The response is forwarded on the upstream port at 480Mb/s. This two-part, start request and complete request mechanism does add a little control complexity to a high-speed bus but it preserves most of the available bandwidth for high-speed devices that need it.

Figure 14.10 A full-speed read on a high-speed bus.

Communicating with the low-speed I/O device on port 3 uses the same mechanism as the full-speed case, with the addition of a preamble token. The low-speed bus is running 320 times slower than the high speed bus but requests and responses travel at 480Mb/s when on the high-speed bus due to the store-and –forward action of the mini host controller. This is a better system solution.

I added another full-speed device on port 4 to highlight an implementation option of a high/full/slow-speed hub. The hub designer may choose to implement a single mini host controller per hub or may choose to implement a mini host controller per downstream port. A single implementation is cheaper while a multiple implementation will be higher performance. In the case of a multiple implementation, each mini host controller will use the same store and forward mechanisms on this port also AND it will allocate another 12Mb/s of bandwidth. You have, in effect, another full/low-speed segment. Yes, if you connected a full-speed device to each of the downstream ports then each would get its own 12Mb/s of bandwidth. So those data-hog USB 1.1 cameras that you bought, you can now connect them all with this style of USB 2.0 compliant hub.

Figure 14.11 shows a typical system with a mixture of “new” and “old” hubs. The good news is that data bandwidths are increased almost everywhere.

Figure 14.11 Increased bandwidths on USB segments.

BUILDING A COMPOUND DEVICE

Now that we understand the construction and operation of a hub, let's look in detail at the design of a hub with I/O functionality, or at an I/O device with hub functionality—both are equivalent descriptions of a compound device. Figure 14-12 looks inside a typical compound device.

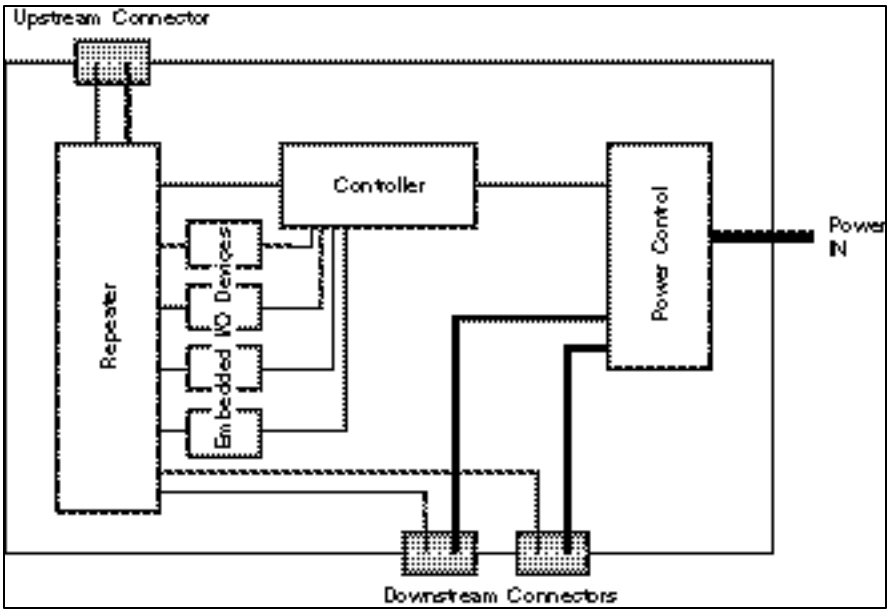


Figure 14-12. A compound device has embedded ports

The number of "external" downstream ports and the number of "internal" downstream ports is ours to decide. An external downstream port is exactly what you would find on a basic hub. Embedded I/O functions are implemented on internal downstream ports. The PC host is unaware of this "external" and "internal" hardware implementation—it just sees devices connected to a hub. A compound device is a hardware optimization that gives the hub controller a little more work to do but saves us the cost of a USB interface and separate controller.

DESIGN EXAMPLE

A convenient place to put a USB hub is inside the monitor of a PC system. An alternative "computer" place is the keyboard, but I prefer the monitor because it has mains power connected that allows the hub to be self-powered. Many other examples could be chosen. I'll use a monitor example as a vehicle to explain the design but will be as flexible as possible so you can fit the concepts into your application.

To make this example a little more interesting, I am going to design a range of monitors as shown in Figure 14-13. The entry system is just a monitor—it has a VGA connector on the back and a set of buttons that control the adjustment of the monitor picture (size, position, pincushion) in a simple base. The models get more elaborate as I add more features—the first has a USB hub in the base. Then I add a combination of features to produce higher-end products—features such as an IR data transfer port, ports for a keyboard and mouse, speakers, microphone, perhaps even parallel and serial ports and a smart card reader. The flexibility of the solution allows easy configuration of any combination of features.

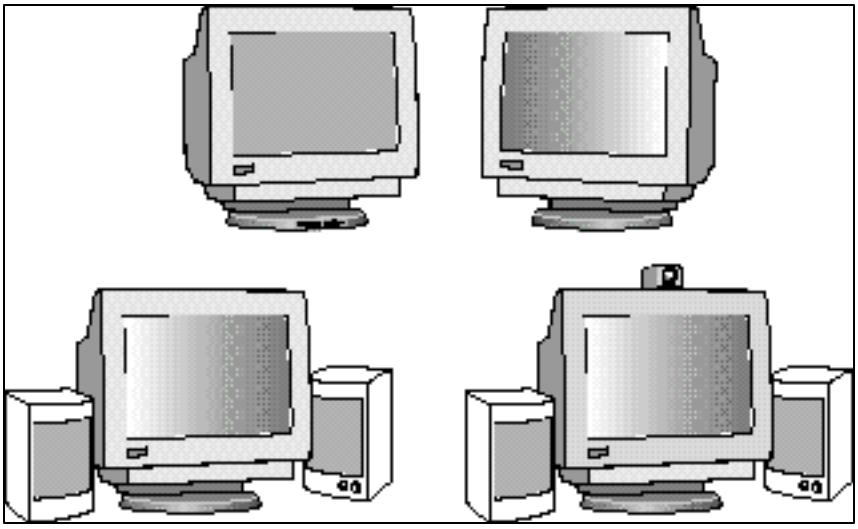


Figure 14-13. A range of monitor solutions

Figure 14-14 shows the internal block diagram of a typical monitor with its connection to the PC. It is microcontroller-based with an I2C bus interconnecting the building blocks.

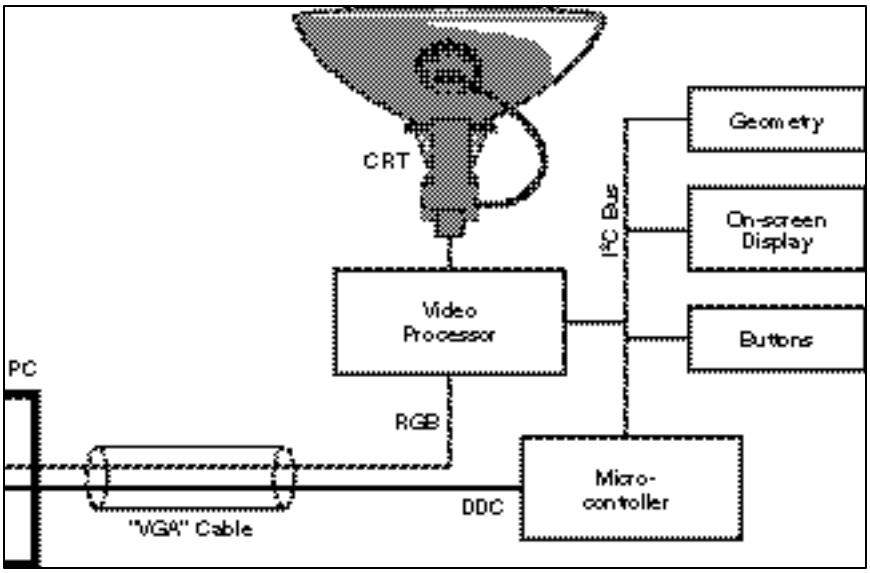


Figure 14-14. Block diagram of a typical monitor

The connection to the PC is minimally a VGA connection. Monitor manufacturers typically implement a Display Data Channel (DDC) using an I2C bus in the same cable that is used in their manufacturing group for monitor alignment. The lack of a standard Windows driver for this connection has hampered its inclusion on standard video cards, so, for the foreseeable future, monitors will continue to have separate alignment buttons. Our example design implements these buttons in an exchangeable base unit.

There are many options for adding a USB hub to this example, and each involves different tradeoffs.

Step 1: Adding a Hub

If we just want to include a basic hub, then a ready-built hub with I2C connection, such as a Texas Instruments TUSB2140 or a Philips PDIUSBH11A, would be a good choice (Figure 14-15). These components appear as slave I2C peripherals to the monitor microcontroller, and we would need to add minimal firmware to the monitor microcontroller to support them.

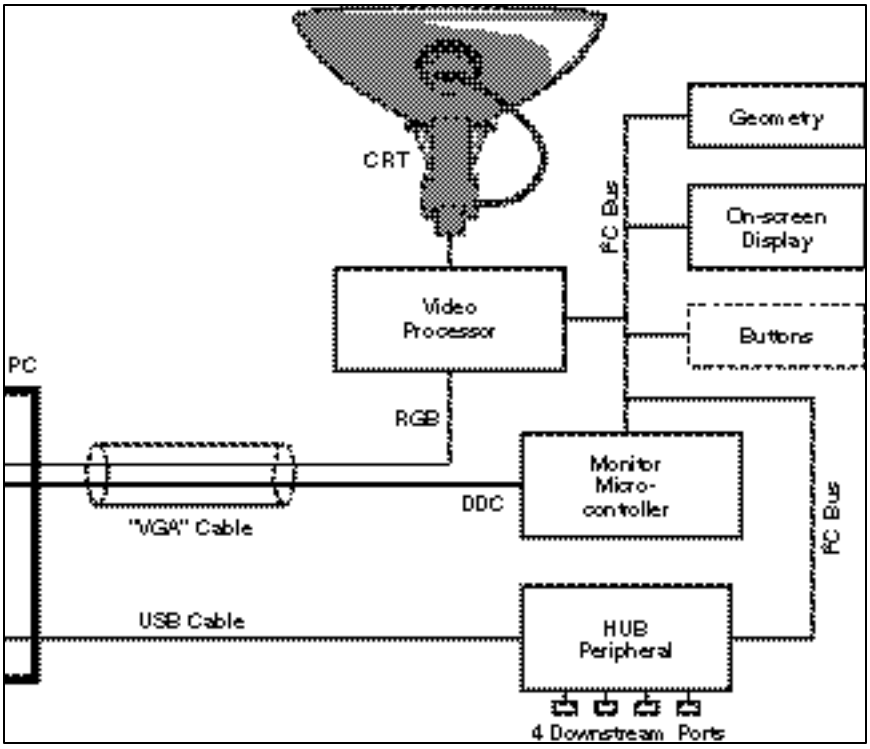


Figure 14-15. Using an I2C connected hub

The I^2C bus from the USB hub peripheral component is addressable via USB. Low-bandwidth data can be transferred between the PC host and the monitor microcontroller using this I^2C embedded I/O function.

We'll replace the physical alignment buttons on the monitor with software on the PC host. The command set required to implement picture adjustment using I^2C has been standardized by a VESA working group (VESA = Video Electronics Standards Group at www.VESA.org), and the document is included in the Chapter 14/VESA directory on the CD-ROM. Figure 14-16 shows a demonstration program written by STMicroelectronics that sends these picture adjustment commands via USB and the hub's I^2C bus to the monitor microcontroller. When the monitor microcontroller powers up, it looks for a "button-base" or a "hub-base" on its I^2C bus and operates accordingly.

We have replaced the cost of the buttons with the cost of the hub circuitry; to put it another way, we got a USB hub for free!

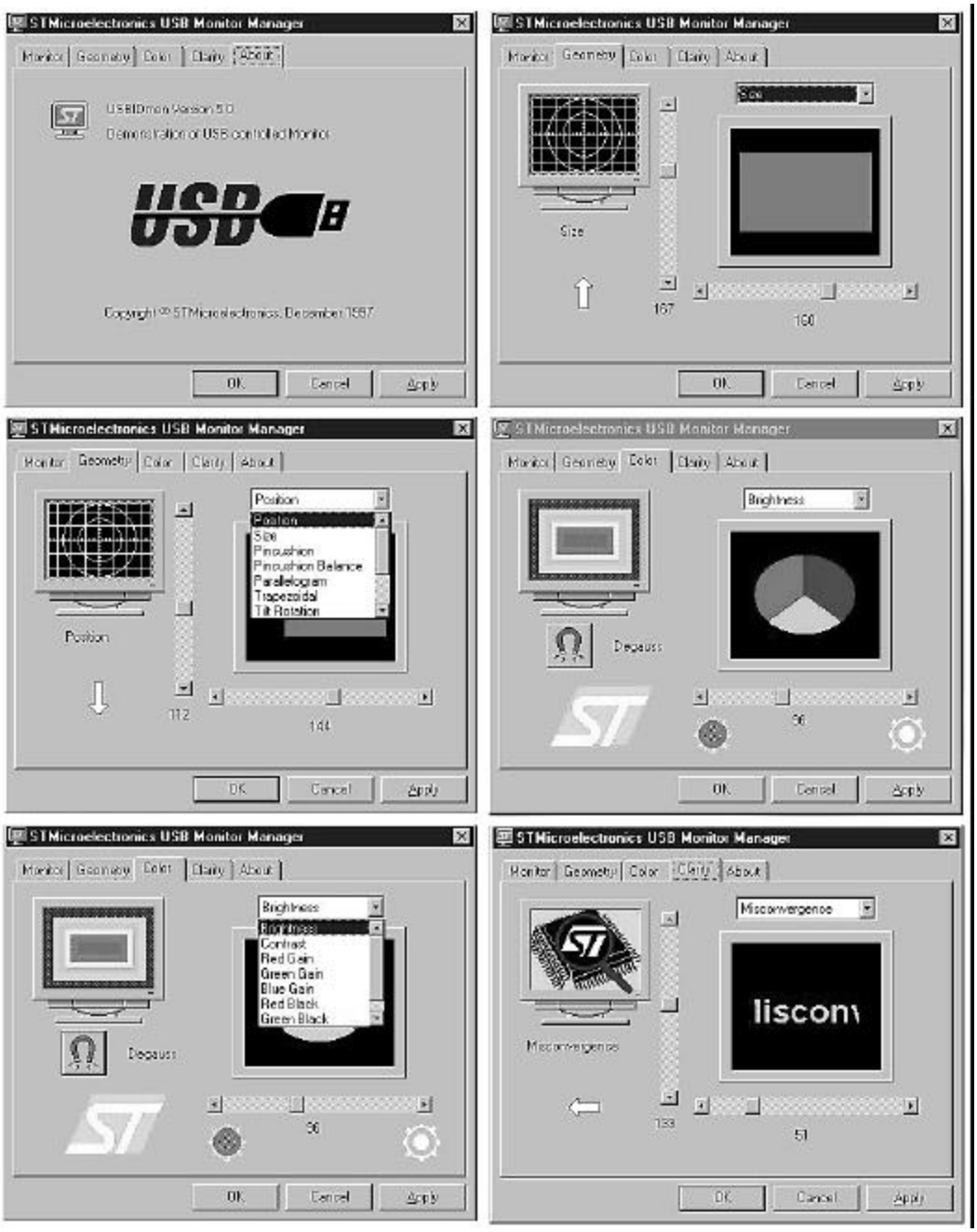


Figure 14-16. An application controls monitor alignment

Step 2: Adding I/O Devices

We are going to design modular I/O functionality in the monitor, so a hub component with an integrated microcontroller and more endpoints (Figure 14-17) would be a better choice. We'll discover that the "alignment button" application and the "hub controller" application use little of the capabilities of the hub microcontroller. To look at it another way, we have ample headroom to add functionality without adding high cost to the solution.

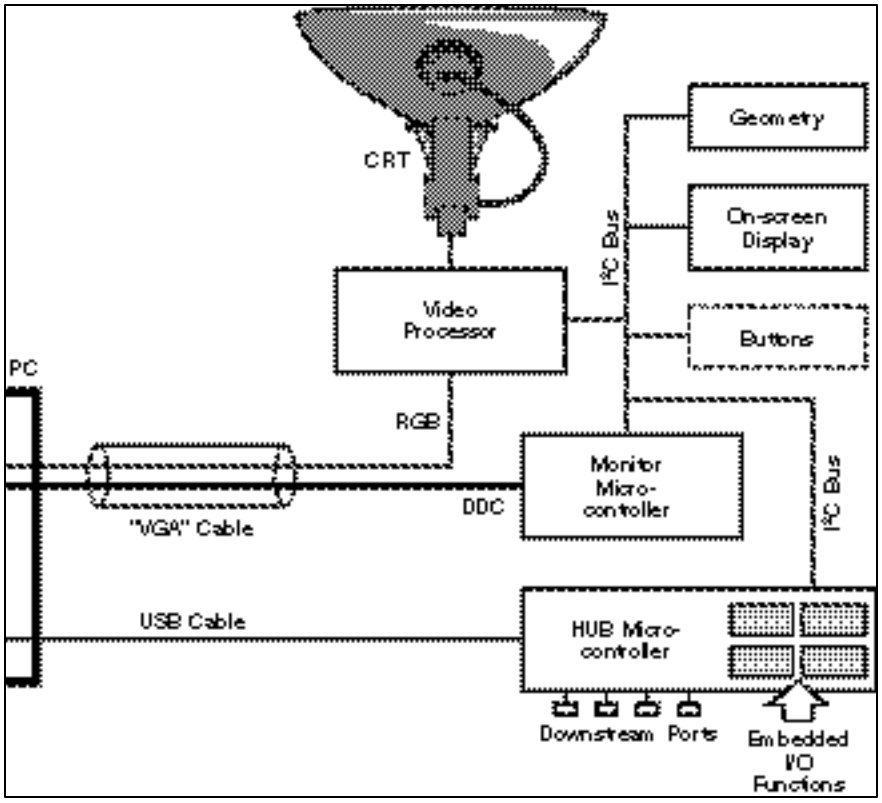


Figure 14-17. Adding I/O flexibility with a microcontroller

Step 3: Extensible Design

Figure 14-18 shows an example of a high-end monitor implementation. The left side of the diagram shows the hardware and the right side shows the software.

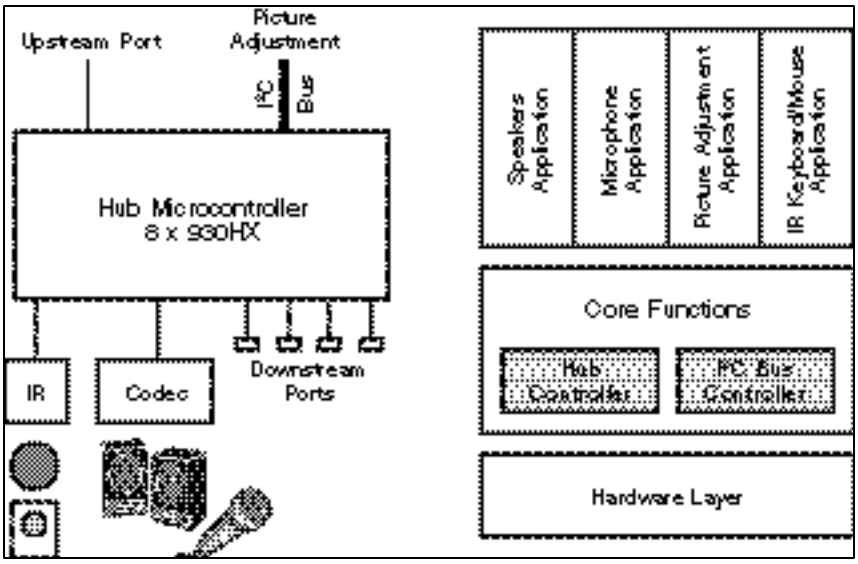
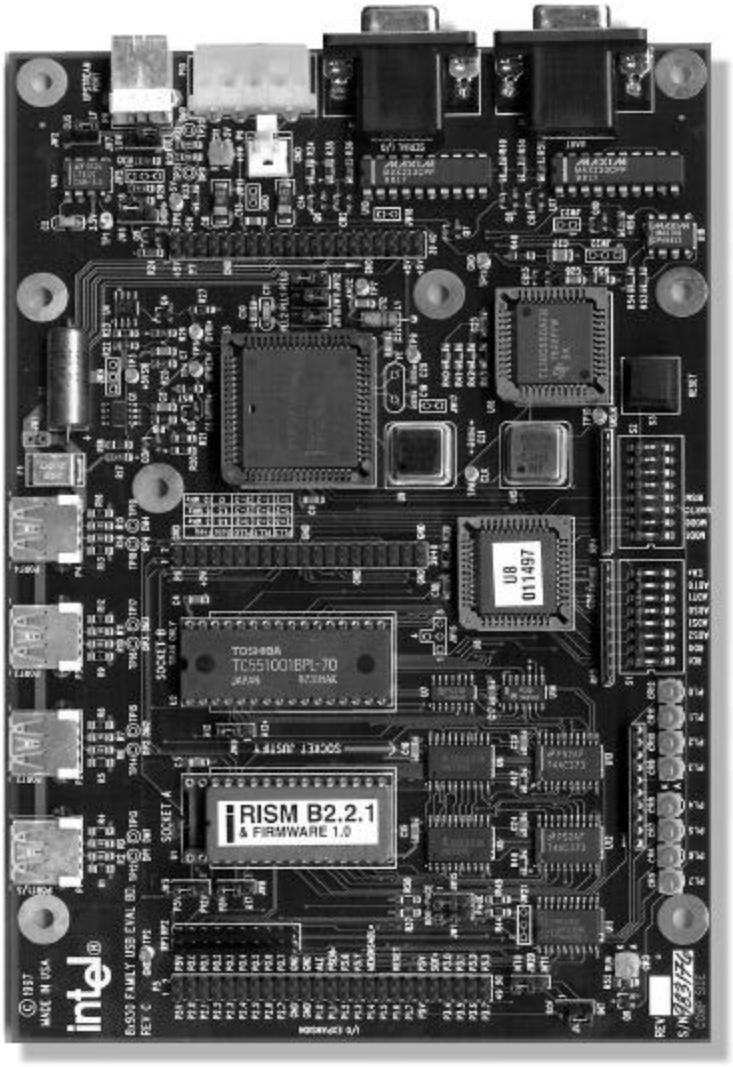


Figure 14-18. High-end USB monitor example

The core firmware to manage the hub and embedded function capabilities is included on the CD-ROM. Descriptors and applications code to match the I/O hardware is added on top of this core code in a building-block fashion. Each I/O function is described by a unique interface descriptor that specifies independent endpoints. All of these interfaces will run concurrently. The example includes a four-port hub, picture adjustment controls via the I^2C bus, speakers and microphone using asynchronous endpoints, and an IR data transfer port. There is still capability with the Intel 8x930HX component (three unused endpoints) that would allow adding more functions such as a printer port, serial ports, mouse, keyboard, floppy disk, or a smart card reader.

A videoconferencing camera needs its own special-purpose controller to implement data compression, so this could not be added as an application on top of the core firmware. A camera could be attached to one of the downstream hub ports. All these features and we have only one USB connection back to the PC host! Figure 14-19 shows a photograph of the completed example design.



Courtesy of Intel Corp.

Figure 14-19. Incremental features for a monitor solution

CHAPTER SUMMARY

This chapter has shown that a hub design is straightforward. Several manufacturers build plug-and-go hub components that are hard-coded to operate in a USB environment. Care must be taken on the power management and overcurrent protection, and detailed design recommendations are included on the CD-ROM. Some hub components use an embedded or external microcontroller to implement more embedded I/O functionality—the firmware to implement the base functionality is often provided by the component manufacturer.

Embedded I/O devices are easy to add to a hub controller that supports multiple device endpoints. The design methodology is the same as that for a stand-alone I/O device. Modules of code to support each interface can even be moved from a stand-alone I/O device into an embedded device supported by the same microcontroller.

I/O devices embedded in a hub give the USB designer an additional degree of freedom when designing a system. And, as we have seen, the design is not difficult.